

**UNITED STATES PATENT APPLICATION**

**DECOUPLED VECTOR ARCHITECTURE**

**INVENTORS**

**GREGORY J. FAANES**  
**STEVEN L. SCOTT**  
**ERIC P. LUNDBERG**  
all of Eau Claire, Wisconsin

**WILLIAM T. MOORE, JR.**  
of Elk Mound, Wisconsin

**TIMOTHY JOHNSON**  
of \_\_\_\_\_, Wisconsin

Schwegman, Lundberg, Woessner, & Kluth, P.A.  
1600 TCF Tower  
121 South Eighth Street  
Minneapolis, Minnesota 55402  
ATTORNEY DOCKET 1376.699US1

## DECOUPLED VECTOR ARCHITECTURE

### Related Applications

This application is related to U.S. Patent Application No. \_\_\_\_\_,  
5 entitled "Multistream Processing System and Method", filed on even date herewith; to  
U.S. Patent Application No. \_\_\_\_\_, entitled "System and Method for  
Synchronizing Memory Transfers", Serial No. \_\_\_\_\_, filed on even date  
herewith; to U.S. Patent Application No. \_\_\_\_\_, entitled "Decoupled Store  
Address and Data in a Multiprocessor System", filed on even date herewith; to U.S.  
10 Patent Application No. \_\_\_\_\_, entitled "Latency Tolerant Distributed  
Shared Memory Multiprocessor Computer", filed on even date herewith; to U.S. Patent  
Application No. \_\_\_\_\_, entitled "Relaxed Memory Consistency Model",  
filed on even date herewith; to U.S. Patent Application No. \_\_\_\_\_, entitled  
"Remote Translation Mechanism for a Multinode System", filed on even date herewith;  
15 and to U.S. Patent Application No. \_\_\_\_\_, entitled "Method and  
Apparatus for Local Synchronizations in a Vector Processor System", filed on even date  
herewith, each of which is incorporated herein by reference.

### Field of the Invention

20 The present invention is related to vector processing computers, and more  
particularly to systems and methods for hiding memory latency in a vector processing  
computer.

### Background Information

25 As processors run at faster speeds, memory latency on accesses to memory  
looms as a large problem. Commercially available microprocessors have addressed this  
problem by decoupling address computation of a memory reference from the memory  
reference itself. In addition, the processors decouple memory references from execution  
based on those references.

The memory latency problem is even more critical when it comes to vector processing. Vector processors often transfer large amounts of data between memory and processor. In addition, each vector processing node typically has two or more processing units. One of the units is typically a scalar unit. Another unit is a vector execution unit. In the past, the scalar, vector load/store and vector execution units were coupled together in order to avoid memory conflicts between the units. It has been, therefore, difficult to extend the decoupling mechanisms of the commercially available microprocessors to vector processing computers.

What is needed is a system and method for hiding memory latency in a vector processor that limits the coupling between the scalar, vector load/store and vector execution units.

### **Brief Description of the Drawings**

Fig. 1 illustrates a vector processing computer according to the present invention;

Fig. 2 illustrates an alternate embodiment of a vector processing computer according to the present invention;

Fig. 3 illustrates yet another embodiment of a vector processing computer according to the present invention;

Fig. 4 illustrates a multiprocessing computer system according to the present invention;

Fig. 5 illustrates a node which could be used in the multiprocessor computer system of Fig. 4;

Fig. 6 illustrates a processor which could be used in the node of Fig. 5;

Fig. 7 is a more detailed description of a processor which could be used in the node of Fig. 5; and

Figs. 8a-8c illustrate instruction flow for vector instructions according to the present invention.

### **Description of the Preferred Embodiments**

In the following detailed description of the preferred embodiments, reference is made to the accompanying drawings which form a part hereof, and in which is shown by way of illustration specific embodiments in which the invention may be practiced. It is to be understood that other embodiments may be utilized and structural changes may be made without departing from the scope of the present invention.

As noted above, the coupling together of the various units of a vector processor has made it difficult to extend the decoupling mechanisms used in commercially available microprocessors to vector processing computers. Standard scalar processors often decouple the address computation of scalar memory references and also decouple scalar memory references from scalar execution. The MIPS R10K processor is one example of this approach. A method of extending these concepts to a vector processor is discussed below.

A vector processing computer 10 is shown in Fig. 1. Vector processing computer 10 includes a scalar processing unit 12, a vector processing unit 14 and a memory 16. In the example shown, scalar processing unit 12 and vector processing unit 14 are connected to memory 16 across an interconnect network 18. In one embodiment, vector processing unit 14 includes a vector execution unit 20 connected to a vector load/store unit 22. Vector load/store unit 22 handles memory transfers between vector processing unit 14 and memory 16.

The vector and scalar units in vector processing computer 10 are decoupled, meaning that scalar unit 12 can run ahead of vector unit 14, resolving control flow and doing address arithmetic. In addition, in one embodiment, computer 10 includes load buffers. Load buffers allow hardware renaming of load register targets, so that multiple loads to the same architectural register may be in flight simultaneously. By pairing vector/scalar unit decoupling with load buffers, the hardware can dynamically unroll loops and get loads started for multiple iterations. This can be done without using extra architectural registers or instruction cache space (as is done with software unrolling and/or software pipelining).

In one embodiment, both scalar processing unit 12 and vector processing unit 14 employ memory/execution decoupling. Scalar and vector loads are issued as soon as possible after dispatch. Instructions that depend upon load values are dispatched to queues, where they await the arrival of the load data. Store addresses are computed early  
5 (in program order interleaving with the loads), and their addresses saved for later use.

In one embodiment, each scalar processing unit 12 is capable of decoding and dispatching one vector instruction (and accompanying scalar operand) per cycle. Instructions are sent in order to the vector processing units 14, and any necessary scalar operands are sent later after the vector instructions have flowed through the scalar unit's  
10 integer or floating point pipeline and read the specified registers. Vector instructions are not sent speculatively; that is, the flow control and any previous trap conditions are resolved before sending the instructions to vector processing unit 14.

The vector processing unit renames loads only (into the load buffers). Vector operations are queued, awaiting operand availability, and issue in order. No vector  
15 operation is issued until all previous vector memory operations are known to have completed without trapping (and as stated above, vector instructions are not even dispatched to the vector unit until all previous scalar instructions are past the trap point). Therefore, vector operations can modify architectural state when they execute; they never have to be rolled back, as do the scalar instructions.

20 In one embodiment, scalar processing unit 12 is designed to allow it to communicate with vector load/store unit 22 and vector execution unit 20 asynchronously. This is accomplished by having scalar operand and vector instruction queues between the scalar and vector units. Scalar and vector instructions are dispatched to certain instruction queues depending on the instruction type. Pure scalar  
25 instructions are just dispatched to the scalar queues where they are executed out of order. Vector instructions that require scalar operands are dispatched to both vector and scalar instruction queues. These instructions are executed in the scalar unit. They place scalar operands required for vector execution in the scalar operand queues that are

between the scalar and vector units. This allows scalar address calculations that are required for vector execution to complete independently of vector execution.

5       The vector processing unit is designed to allow vector load/store instructions to execute decoupled from vector execute unit 20. The vector load/store unit 22 issues and executes vector memory references when it has received the instruction and memory operands from scalar processing unit 12. Vector load/store unit 22 executes independently from vector execute unit 20 and uses load buffers in vector execute unit 20 as a staging area for memory load data. Vector execute unit 20 issues vector memory and vector operations from instructions that it receives from scalar processing unit 12.

10       When vector execution unit 20 issues a memory load instruction, it pulls the load data from the load buffers that were loaded by vector load/store unit 22. This allows vector execution unit 20 to operate without stalls due to having to wait for load data to return from main memory 16.

15       Vector stores execute in both the vector load/store unit 22 and the vector execute unit 20. The store addresses are generated in the vector load/store unit 22 independently of the store data being available. The store addresses are sent to memory 16 without the vector store data. When the store data is generated in vector execute unit 20, the store data is sent to memory 22 where it is paired up with the store address.

20       The current approach allows scalar computation to execute independently of vector computation. This allows the scalar address and operand computation to execute in parallel, or ahead of the vector instruction stream. In addition, this approach allows vector address generation to execute independently of vector execution. Finally, this approach allows vector memory operations to execute ahead of vector execution, thus reducing memory latency.

25       This decoupling of the scalar, vector load/store, and vector execution does, however, make synchronizing of scalar and vector data more complex. Computer 10 incorporates specific instructions and hardware that reduce this complexity.

      An alternate embodiment of vector processing computer 10 is shown in Fig. 2. Vector processing computer 10 in Fig. 2 includes a processor 28. Processor 28 includes

a scalar processing unit 12 and two vector processing units (14.0 and 14.1). Scalar processing unit 12 and the two vector processing units 14 are connected to memory 16 across interconnect network 18. In the embodiment shown, memory 16 is configured as cache 24 and distributed global memory 26. Vector processing units 14 include a vector execution unit 20 connected to a vector load/store unit 22. Vector load/store unit 22 handles memory transfers between vector processing unit 14 and memory 16.

In one embodiment, four processors 28 and four caches 24 are configured as a Multi-Streaming Processor (MSP) 30. An example of such an embodiment is shown in Fig. 3. In one such embodiment, each scalar processing unit 12 delivers a peak of 0.4 GFLOPS and 0.8 GIPS at the target frequency of 400 MHz. Each processor 28 contains two vector pipes, running at 800 MHz, providing 3.2 GFLOPS for 64-bit operations and 6.4 GFLOPS for 32-bit operations. The MSP 30 thus provides a total of 3.2 GIPS and 12.8/25.6 GFLOPS. Each processor 28 contains a small Dcache used for scalar references only. A 2 MB Ecache 24 is shared by all the processors 28 in MSP 30 and used for both scalar and vector data. In one embodiment processor 28 and cache 24 are packaged as separate chips (termed the "P" chip and "E" chips, respectively).

In one embodiment, signaling between processor 28 and cache 24 runs at 400 Mb/s on processor-cache connection 32. Each processor to cache connection 32 shown in Fig. 3 uses an incoming 64-bit path for load data and an outgoing 64-bit path for requests and store data. Loads can achieve a maximum transfer rate of 51 GB/s from cache 24. Stores can achieve up to 41 GB/s for stride-one and 25 GB/s for non-unit stride stores.

In one embodiment, global memory 26 is distributed to each MSP 30 as local memory 48. Each Ecache 24 has four ports 34 to M chip 42 (and through M chip 42 to local memory 48 and to network 38). In one embodiment, ports 34 are 16 data bits in each direction. MSP 30 has a total of 25.6 GB/s load bandwidth and 12.8-20.5 GB/s store bandwidth (depending upon stride) to local memory.

In one embodiment, each system 10 includes a plurality of nodes 36 interconnected with a network 38. In one embodiment, as is shown in Fig. 4, network

38 includes high-speed links 40 used to connect nodes 36 either as a hypercube or as a torus. Other approaches for interconnecting nodes 36 could be used as well.

In one embodiment, each node 36 consists of four MSPs 30 and sixteen M chips 42, as shown in Fig. 5. M chips 42 contain memory controllers, network interfaces and cache coherence directories with their associated protocol engines. In one such embodiment, memory 26 is distributed round-robin by 32-byte cache lines across the sixteen M chips 42 at each node 36. Thus, the M chip for a particular address is selected by bits 8..5 of the physical address.

Each E chip is responsible for one fourth of the physical address space, determined by bits 5 and 6 of the physical address. A reference to a particular line of memory is sent to the associated E chip where the Ecache is consulted, and either the line is found in the Ecache or the request is sent on to an M chip. Bits 7 and 8 of the physical address select one of four M chips connected to each E chip.

Each M chip 42 resides in one of sixteen independent slices of the machine, and the interconnection network 38 provides connectivity only between corresponding M chips on different nodes (thus there are sixteen parallel, independent networks). All activity (cache, memory, network) relating to a line of memory stays within the corresponding system slice.

Each M chip 42 contains two network ports 44; each 1.6 GB/s peak per direction. This provides a total peak network bandwidth of 51.2 GB/s in and 51.2 GB/s out. Single transfers to/from any single remote destination will use only half this bandwidth, as only one of two ports 44 per M chip 42 will be used. Also, contention from the other processors 28 on node 36 must be considered. Lastly, all inter-node data is packetized, resulting in a smaller ratio of sustained to peak than in the local memory subsystem. Protocol overheads vary from 33% (one way, stride-1 reads) to 83% (symmetric, non-unit-stride reads or writes).

Each node 36 also contains two I/O controller chips 46 ("I" chips) that provide connectivity between the outside world and network 38 and memory 26. In one embodiment, each "I" chip 46 provides two XIO (a.k.a. Crosstalk) I/O channels 49, with

a peak speed bandwidth of 1.2 GB/s full duplex each. The I chips are connected to each other and to the sixteen M chips 36 with enough bandwidth to match the four XIO channels.

5 This partitioning provides low latency and high bandwidth to local memory 48. With a local memory size of up to 16 GB (64 GB once 1 Gbit chips become available), most single-processor and autotasked codes should run locally, and most references in distributed-memory codes will be satisfied locally as well. Latency to remote memory will depend upon the distance to the remote node, and the level of contention in network 38.

10 In one embodiment, a limited operating system executes on each node, with a Unicos/mk-like layer across nodes 30. The limited OS will provide basic kernel services and management of two direct-attached I/O devices (a disk array and network interface). All other I/O connectivity is provided by a separate host system. In one such embodiment, the host system also provides the user environment (shell, cross compilers, 15 utility programs, etc.), and can be used to run scalar compute applications.

One embodiment of processor 28 is shown in Fig. 6. In the embodiment shown, processor 28 includes three sections: a scalar section (SS) 50, a vector section (VS) 52 and an ecache interface unit (EIU) 54.

20 EIU 54 is the cache/network interface; it operates to ensure high bandwidth between the SS/VS and cache 24.

A more detailed example of one embodiment of processor 28 is shown in Fig. 7. In the embodiment shown in Fig. 7, SS 50 includes a *Instruction Fetch Unit* (IFU) 80, a *Dispatch Unit* (DU) 82, an AS Unit (ASU) 84, a *Load/Store Unit* (LSU) 86, and a *Control Unit* (CU) 88. AS Unit 84 includes A Unit 90 and S Unit 92. In one such 25 embodiment, SS 50 implements an out-of-order, 2-way issue superscalar processor.

VS 52 implements a two pipe vector processor capable of executing eight floating-point operations and 4 memory operations per Lclk. VS 52 includes a *Vector Dispatch Unit* (VDU) 64, a *Vector Unit* (VU) 66 and a *Vector Load/Store Unit* (VLSU) 68.

In one embodiment, SS 50 is a high performance superscalar processor that implements many of today's RISC superscalar features. It can dispatch, in-order, up to two instructions per Lclk, can then execute instructions out-of-order within the various units, and then graduate in-order up to two instructions per Lclk. The SS also  
5 implements speculative execution, register renaming, and branch prediction to allow greater out-of-order execution. The SS can predict up to two branch instruction and uses a *Branch History Table ( BHT )*, a *Jump Target Buffer ( JTB )*, and *Jump Return Stack ( JRS )* to help insure a high branch prediction rate.

SS 50 also contains two 64-bit wide register files, *A Registers ( AR )* and *S  
10 Registers ( SR )* 58. The ARs are used mainly for address generation. In one embodiment, there are 64 logical ARs and 512 physical ARs that are renamed within A Unit 90. The SRs are used for both integer and floating-point operations. In one embodiment, there are 64 logical SRs and 512 renamed physical SRs within S Unit 92.

As noted above, SS 50 is capable of issuing up to two integer operations per  
15 Lclk using the ARs. In addition, SS 50 is capable of issuing one SR instruction per Lclk that can be either integer or floating point. The decoupled LSU of the SS can issue, in order, one load or store per Lclk which may then execute out-of-order with respect to previous scalar memory operations. The SS unit is also able to issue one branch instruction per Lclk, which allows one branch prediction to be resolved per Lclk.

20 The SS includes separate first level of caches for instructions and scalar data. SS *Instruction Cache ( Icache )* 60 is 16KBytes in size and is two-way set associative. Each Icache line is 32 Bytes. *Data Cache ( Dcache)* 62 is also 16KBytes and two-way set associative with a 32 Byte line size. Icache 60 is virtually indexed and virtually tagged while Dcache 62 is virtually indexed and physically tagged. In one embodiment,  
25 Dcache 62 is write through.

SS 50 supports virtual memory addressing by maintaining precise exceptions for address translation errors on both scalar and vector memory references. Floating-point exceptions and other errors are not precise.

VS 52 is a two pipe vector processor that executes operations at the Sclk rate. Control for VS 52 executes at the LClk rate. In one embodiment, VS 52 consists of three units, VDU 64, VU 66 and VLSU 68, that are all decoupled from SS 50 and are also decoupled from each other. VDU 64 receives instruction from the DU of SS 50 and  
5 couples required scalar operands with these vector operations. The VDU then dispatches these vector instructions to the VU and VLSU. This allows scalar address generation for vector memory references to execute ahead of the vector operations and also permits the VLSU to run ahead of the VU. This decoupling, similar to modern RISC CPUs, helps VS 52 conceal latency to both Ecache 24 and memory system 26.

10 VU 52 contains the large *Vector Register ( VR )* file 70, which can operate in two widths, 32-bits or 64-bits. There are 32 logical and 32 physical VRs 70 that have a *Maximum Vector Length ( MaxVL )* of 64 elements. No register renaming is performed on the VRs. When performing 32-bit operations, the VRs are 32-bits wide and when executing 64-bit operations the VRs are 64-bits wide. For both widths of operations, the  
15 MaxVL of the VRs remains at 64 elements. The VU also contains a *Vector Carry Register ( VC )* that is 1-bit wide and MaxVL elements long. This register is used to support extended integer arithmetic.

The VU contains two other register sets that are used to control the vector operations of both the VU and VLSU. A *Vector Length Register ( VL )* 72 is a single 64-  
20 bit register that contains a value between 0 and MaxVL. The VL serves to limit the number of operations a vector instruction performs. The VS also includes eight *Vector Mask Registers ( VM )* 74 which are each 1-bit wide and MaxVL elements long. VMs 74 are used to control vector instructions on a per element basis and can also contain the results of vector comparison operations.

25 VU 66 is capable of issuing, in order, one vector operation per Lclk. The VU contains two copies, one for each vector pipe, of three *Functional Groups ( FUG s )* 76, that are each capable of executing one operation per Sclk. Each FUG 76 consists of multiple functional units that share data paths to and from VRs 70. The FUGs 76 contain both integer and floating-point function units. Vector chaining is supported

between the FUGs and writes to memory 26 but is not supported for reads from memory 26. Most 32-bit vector operations will be performed at twice the rate of 64-bit vector operations.

5 VLSU 68 performs all memory operations for VS 52 and is decoupled from VU 66 for most operations. This allows VLSU 68 to load vector memory data before it is used in the VU. The VLSU can issue, in order, one instruction per Lclk. The VLSU can generate four memory operations per Lclk. In one embodiment, the four addresses can be for stride loads, gathers, stride stores, or scatters.

To support decoupling of VLSU 68 from VU 66, VS 52 also contains a large  
10 *Load Buffer (LB)* 78. In one embodiment, LB 78 is a 512 64-bit element buffer. That is, there are eight vector-register-equivalent buffers that can be used as a staging area for vector loads. Memory load data is placed in LB 78 by VLSU 68 until VU 66 has determined that no previous memory operations will fault. VU 66 then moves the load data from LB 78 into the appropriate VR 70. In one embodiment, VLSU 68 rotates  
15 through the eight load buffers 78 as needed. On a Vector Translation Trap, the load buffer contents are ignored and the vector load instructions associated with the data in the load buffers are restarted.

In one embodiment, AU 90 includes a load buffer used to decouple a scalar load from a scalar operation. In one embodiment, SU 92 includes a load buffer used to  
20 decouple a scalar load from a scalar operation.

In one embodiment, VS 52 includes support for vector register renaming in a manner similar to that discussed for SS 50 above.

VS also supports virtual memory addressing by maintaining precise exception state for vector address translation errors.

25 EIU 54 maintains a high bandwidth between a P Chip 28 and E Chips 24. In one embodiment, EIU 54 is capable of generating a maximum of four memory references per Lclk to the E Chips 24 and can also receive up to four Dwords of data back from the E Chips 24 per Lclk.

EIU 54 supplies instructions to Icache 60 that are fetched from E Chips 24. EIU 54 also controls all scalar and vector memory requests that proceed to E Chips 24. The EIU contains buffering that allows multiple scalar and vector references to be queued to handle conflicts when multiple references request the same E Chip in a single LClk.

5 EIU 54 also cooperates with DU 82, LSU 86 and VLSU 68 to ensure proper local synchronization of scalar and vector references. This requires that EIU 54 have the capability to invalidate vector store addresses that are resident in Dcache 62.

### Instruction Flow

10 All instructions flow through the processor 28 by first being fetched by IFU 80 and then sent to DU 82 for distribution. In the DU, instructions are decoded, renamed and entered in order into Active List (AL) 94 at the same time that they are dispatched to AU 90, SU 92 and VDU 64.

In one embodiment, a 64 entry AL 94 keeps a list of currently executing  
15 instructions in program order, maintaining order when needed between decoupled execution units. Instructions can enter AL 94 in a speculative state - either branch speculative or trap speculative or both. A branch speculative instruction is conditioned on a previous branch prediction. An instruction is trap speculative if it or any previous instruction may trap. Speculative instructions may execute, but their execution cannot  
20 cause permanent processor state changes while speculative.

Instructions in the AL proceed from *speculative* to *scalar committed*, to *committed*, to *graduated*. The AL uses two bits to manage instruction commitment and graduation: *trap* and *complete*. An instruction that may trap enters the AL with its trap bit set and complete bit clear. The trap bit is later cleared when it is known the  
25 instruction will not trap. Both the trap and complete bits are set for a trapping instruction.

A *scalar committed* instruction is not branch speculative, will not generate a scalar trap, and all previous *scalar* instructions will not trap. Scalar commitment is

needed for the *Vector Dispatch Unit ( VDU )* to dispatch vector instructions to VU and VLSU. Instructions proceed from being scalar committed to committed.

5        *Committed* instructions are not branch speculative, will not trap, and *all* previous instructions will not trap. After commitment, an instruction proceeds to *graduation* and is removed from the AL. An instruction may only graduate if it and all previous  
10 instructions in the AL are marked complete and did not trap. In one embodiment, the AL logic can graduate up to four instructions per LClk. Scalar shadow registers are freed at graduation and traps are asserted at the graduation point. Instructions cannot be marked complete until it can be removed from the AL. That requires at the least that all  
15 trap conditions are known, all scalar operands are read, and any scalar result is written. Some vector instructions are marked complete at dispatch, others at vector dispatch, and the remaining vector instructions are marked complete when they pass vector address translation.

Scalar instructions are *dispatched* by DU 82 in program order to AU 90 and/or  
15 to SU 92. Most scalar instructions in the AU and SU are *issued* out-of-order. They *read* the AR or SR, *execute* the indicated operations, *write* the AR or SR and send instruction completion notice back to the DU. The DU then marks the instruction *complete* and can *graduate* the scalar instruction when it is the oldest instruction in the AL.

20        All scalar memory instructions are *dispatched* to AU 90. The AU *issues* the memory instructions in-order with respect to other scalar memory instructions, *reads* address operands from AR 56 and sends the instruction and operands to LSU 86. For scalar store operations, the memory instruction is also dispatched to the AU or SU to *read* the write data from the AR or SR and send this data to LSU 86.

25        The LSU performs address translation for the memory operations received from the AU in-order, sends instruction *commitment* notice back to the DU, *executes* independent memory operations out-of-order. For scalar loads, when load data is written into the AR or SR, the AU or SU transmits instruction *completion* notice back to the DU. Scalar store instruction *completion* is sent by the EIU 54 to the DU when the write data has been sent off to cache 24.

Branch instructions are predicted in the IFU before being sent to the DU. The DU *dispatches* the branch instruction to either the AU or SU. The AU or SU *issues* the instruction, *reads* AR or SR, and sends the operand back to the IFU. The IFU determines the actual branch outcome, signals promote or kill to the other units and  
5 sends completion notice back to the DU.

The IFU supplies a stream of instructions to Dispatch Unit (DU) 82. This stream of instructions can be speculative because the IFU will predict the result of branches before the outcome of the branch is resolved. AU 90 and SU 92 communicate with the IFU to resolve any outstanding predicted branches. For branch instructions that are  
10 predicted correctly, the branch instruction is simply promoted by the IFU. For branch mispredictions, the IFU will signal a mispredict to the other units, restart fetching instructions at the correct instruction address and supply the proper instruction stream to the DU.

Instructions can enter the AL speculated on one or two previous branch  
15 predictions. Each instruction is dispatched with a two bit branch mask, one for each branch prediction register. Instructions can be dependent on either or both branch prediction. A new branch prediction is detected at the point in the instruction dispatch stream where a branch mask bit is first asserted. A copy of the AL tail pointer is saved on each new branch prediction. If the branch is later killed, the tail pointer is restored  
20 using the saved copy.

At most one new prediction can be dispatched per clock cycle. All predictions are either promoted or killed by the *Branch Prediction Logic ( BPL )*. There can be two promotions in a LClk, but only one branch kill. If there are two outstanding branches, a kill of the older prediction implies a kill of the younger branch prediction. The tail will  
25 be restored using the saved tail from the older branch prediction, while the saved tail from the younger prediction is discarded.

Another responsibility of the AL logic is to maintain memory consistency. The AL logic orders memory references in the LSU and VLSU, it informs the LSU when scalar memory references have been committed so references can be sent to memory,

and the AL logic communicates with other units to perform memory synchronization instructions. The AL logic uses the AL to perform these tasks.

Because of a potential deadlock, no vector store reference can be sent to memory prior to a scalar or vector load that is earlier in program order. Vector load to vector store ordering is done by a port arbiter in EIU 54 while the AL is used to enforce scalar load to vector store ordering.

A scalar load will have a *reference sent* bit set in the AL if the load hits in the Dcache, or when the port arbiter in the EIU acknowledges the scalar load. The AL has a "reference sent" pointer. This pointer is always between the graduation and scalar commit pointers. The reference sent pointer will advance unless the instruction it points to is a scalar load without its reference sent bit set. When the pointer advances past a vector store instruction, it increments a vector store counter in the VLSU. A non-zero count is required for the VLSU to translate addresses for a vector store reference. The vector store counter is decremented when vector memory issue starts translation of addresses for a vector store instruction.

Scalar loads are initially dispatched in the *may trap* state. If the address generation and translation in the LSU results in either a TLB miss or an address error, the complete bit is set, indicating a trap. If translation passes, the trap bit is cleared. Scalar loads to non-I/O space may be satisfied in the Dcache or make memory references immediately after successful address translation. The instruction will later complete when the memory load data is written to a physical register. Scalar load references may bypass previous scalar stores provided the loads are to different addresses.

Prefetch instructions are initially dispatched to the LSU in the may trap state but never generate a trap. The trap bit is always cleared and complete bit set after address translation in the LSU. If the address translation was unsuccessful, the prefetch instruction acts as a NOP, making no Dcache allocation or memory reference.

Vector instructions are *dispatched* in-order from DU 82 to VDU 64. VDU 64 dispatches vector instructions to both VU 66 and VLSU 68 in two steps. First, all vector

instructions are *vpredispatched* in-order in the VDU after all previous instructions are scalar committed. The VDU separates the stream of vector instructions into two groups of vector instructions, the VU instructions and VLSU instructions. All vector instructions are sent to the VU, but only vector memory instructions and instructions  
5 that write VL and VM sent to the VLSU.

Vector instructions that are not vector memory instructions or require no scalar operands are marked complete by the DU at dispatch. An example of the instruction flow of such a vector instruction is shown in Fig. 8a. This instruction class can then graduate from the active list before the VDU has *vdispatched* them to the VU.

10 For vector instructions that require scalar operands, the DU also *dispatches* the vector instructions to the AU and/or SU. An example of the instruction flow of such a vector instruction is shown in Fig. 8b. The vector instruction is *issued* out-of-order in the AU and/or SU, then *reads* AR and/or SR, and finally sends the scalar operand(s) to the VDU where it is paired with the vector instruction. VDU 64 then *vdispatches* the  
15 instruction and data in-order to the VU. Vector instructions that are not memory instructions but do require scalar operand(s), are marked *complete* in the AL when the scalar operand(s) have been read in the AU and/or SU.

Vdispatch and vlstdispatch are decoupled from each other. For vector instructions that are forwarded to the VLSU, the VDU collects all required scalar  
20 operands and *vlstdispatches* the instruction and data in-order to the VLSU.

Vector memory instructions are dispatched to both the VU and VLSU. An example of the instruction flow of such a vector instruction is shown in Fig. 8c. Vector memory operations *execute* in the VLSU decoupled from the VU and send vector memory instruction *completion* notices back to the DU after vector address translation.

25 If a vector memory instruction results in a translation fault, VLSU 68 asserts a Vector Translation Trap. The instruction in DU 82 reaches the commit point but it won't pass the commit point. It stays there and the rest of the instructions move along and all graduate.

Instructions are not allowed to modify the permanent architectural state until they graduate. Since the instruction that caused the translation fault is not allowed to graduate, you have a clear indication of where to restart. The operating system patches up the memory and the program gets restarted at that point.

5           Errors in the VU are handled differently. The vector execution unit makes all of its references and all of its instructions in order so, if it has a fault, it just stops.

For vector load instructions, the VU *issues* the vector memory instruction in-order after address translation has completed in the VLSU and all load data has returned from the memory system. Next, the VU reads data from the LB and *writes* the elements  
10   in the VR.

For vector store instructions, the VU waits for completion of address translation in the VLSU, *issues* the store instruction, and *reads* the store data from the VR.

For non-memory instructions, the VU *issues* the vector instruction in-order, *reads* VR(s), *executes* the operation in the VU's two vector pipes and *writes* the results  
15   into the VR.

Instructions are dispatched to AU 90, SU 92 and VDU 64 by DU 82. The DU will dispatch one class of instructions to only one unit, other instructions to two units, and other instruction types to all three units.

In one embodiment, DU 82 receives a group of eight instructions from IFU 80  
20   per LClk. Only one of these instructions can be a branch or jump instruction. Some of these instructions in the group may be invalid.

The DU decodes the instruction group, determines register dependencies, renames instructions that require renaming, dispatches up to four instructions per LClk to the AU, SU, and/or VDU, and then tracks the state of the instructions in the execution  
25   units. Outgoing instructions are queued in the AU, SU, and VDU instruction queues. When instructions have completed in the execution units, the DU is informed and the instructions are graduated in order.

Decode Logic (DL) in the DU receives up to eight valid instructions from the IFU per LClk. This instruction group is aligned to a 32-Byte boundary. Some of the

instructions may not be valid because of branch and jump instructions within the group of eight instructions. A control transfer to a instruction within a group of eight, will have the instructions before the target instruction invalid.

5 The DL can decode up to 4 instructions per LClk. These instructions must be naturally aligned on 16-Byte boundaries. The DL will determine, for each instruction, the instruction attributes, instruction register usage, and instruction destinations for dispatch. The DL will then advance this decoded information to the register renaming logic.

10 Renaming Logic (RL) in the DU accepts up to 4 decoded instructions from the DL per LClk. The RL determines instruction register dependencies and maps logical registers to physical registers. The AR and SR operand registers of the instructions in the group are matched against the AR and SR result registers of older instructions. This means that the operands must be compared to older instructions within the instruction group and also to older instructions that have already be renamed in earlier LClks.

15 The RL uses a simpler method for register renaming than true general register renaming. This method is termed *Register Shadowing*. In register shadowing, AU 90 and SU 92 use RAM in their respective units for AR 56 and SR 58, respectfully. The *logical* AR/SR are mapped to the *physical* elements of the RAM to allow 8 specific *physical* elements of the RAM per *logical* AR/SR. This means that logical A5 will be mapped to only one of eight physical registers (A5, A69, A133, A197, A261, A325, 20 A389, A453). Logical A0 and S0 will never be renamed because they are always read as zero.

The mapping of logical registers to physical registers for each AR and SR is maintained by a 3-bit user invisible Shadow Pointer (SP) for each logical register 25 (SP\_A0-SP\_A63, SP\_S0-SP\_S63). At reset all SPs are cleared to zero.

When a AR/SR is used as an operand in a instruction, the SP of that register is used to determine the current physical register. This physical register number is then used as the operand register when the instruction executes in one of the execution units.

When an AR/SR is used as a result register, the SP is incremented by one to point to a new physical register. This physical register will then be written to when the instruction executes in one of the execution units. For any instructions after this instruction that use the same logical register as an operand, it will be mapped to this  
5 new physical register.

To control the renaming of the physical registers, each AR/SR logical register is mapped to a 3-bit user invisible Free Shadow Count (FC) register (FC\_A0-FC\_A63, FC\_S0-FC\_S63). At reset the FC registers are set to seven. For each assignment of a new physical register, the associated FC register is decremented by one. Each time an  
10 instruction is graduated that assigned a new physical register, the associated FC for that logical register is incremented by one. If the RL tries to rename a logical register that has a FC of zero, the RL holds this instruction and all younger instructions in the group until a physical register is free by graduating an instruction that assigned a physical register to that logical register.

15 The RL also controls the remapping of physical registers to logical registers after a branch mispredict. The RL needs to have the capability of restoring the mapping back to the mapping at the time of the branch instruction that was mispredicted. The RL accomplishes this by maintaining two user invisible 3-bit Branch Count registers (BC0, BC1) for each AR and SR. Each BC is associated with one of the two possible  
20 outstanding branch predictions.

The BCs count the number of allocations that have been made to a specific logical register since the predicted branch instruction. For each speculative physical register assignment after the speculative branch instruction, the current BC for that speculative branch and logical register is incremented by one. If a second branch is  
25 encountered before the first branch is resolved, the other 64 BC registers are incremented for register allocation.

When the outcome of a speculative branch is resolved, the SP, FC, BC0 and BC1 registers for all ARs and SRs may be modified. For branches that are predicted correctly, the associated BC is cleared to zero. For branches that are predicted

incorrectly, the FC registers are incremented by the amount in both the BC0 and BC1 registers, the SP registers are decremented by the amount in both the BC0 and BC1 registers, and both BC0 and BC1 registers are cleared to zero. This restores the logical to physical mapping to the condition before the speculative branch.

5           The other circumstance where the RL needs to restore the register mapping is when a precise trap is taken. This could be for scalar or vector loads or stores that have a TLB exception. To recover back to the renamed state of the trapping instruction, the SP registers are set equal to the current contents of the SP registers minus seven plus the associated FC register. The FC registers are then set to seven, and the BC0 and BC1  
10 registers are cleared.

### **Vector Operation**

VDU 64 controls the vector instruction flow to VU 66 and VLSU 68. No branch speculative or scalar speculative vector instructions are sent to the VU or VLSU from  
15 the VDU. Because no scalar speculative vector instructions are sent to the VU and VLSU, the vector units do not have to support a precise trap due to a previous scalar instruction that trapped in SS 50.

The VDU performs five functions. The first role of the VDU is to enforce the scalar commitment of previous scalar instructions. The second action of the VDU is to  
20 separate the vector instruction stream into two streams of instructions: VU instructions and VLSU instructions. The third role of the VDU is to establish the communication required between the VU and VLSU for vector instructions that are sent to both units and demand coupling. The fourth duty of the VDU is to assemble all scalar operands that are required for a vector instruction. The AU and SU will pass these scalar operands  
25 to the VDU. Finally, the VDU dispatches vector instructions to the VU and VLSU. The VDU performs these functions in two stages.

The first stage of the VDU is the Vector PreDispatch Logic (VPDL). The VPDL performs the first three functions of the VDU. Vector instructions are dispatched to the *Vector PreDispatch Queue (VPDQ)* in-order from the *Dispatch Unit (DU)*. The

VPDQ can receive two vector instructions per LClk from DU 82 and can contain up to 32 vector instructions. Each LClk, the VPDL attempts to *vpredispatch* the instruction at the head of the VPDQ to the *Vector Dispatch Queue ( VDQ )* and/or the *Vector Load/Store Dispatch Queue ( VLSDQ )*.

5           Before the VPDL can pass the instruction to the VDQ and/or VLSDQ, all previous scalar instructions must be committed. Scalar commitment notice is sent from the *Active List ( AL )* in the *Dispatch Unit ( DU )*. Each time the scalar commit pointer moves past a vector instruction in the AL, the DU sends the VDU a *go\_vdisp* signal that indicates that it is safe to remove a vector instruction from the VPDQ. VDU can receive  
10       up to four separate *go\_vdisp[0-3]* signals per LClk. The VDU maintains a counter that is incremented each time a *go\_vdisp* signal is received. The VPDL determines that the vector instruction at the head of the VPDQ has no previous scalar speculative instructions if this counter is greater than zero. Each time the VPDL removes a instruction from the VPDQ, it decrements this counter.

15           Because the VU and the VLSU are decoupled from each other, there needs to be a method to ensure that the VLSU does not execute instructions that require vector register operands before the last write of that register in the vector instruction stream in VU. To allow for this coupling between the VU and VLSU, the VDU and VU contain three structures.

20           The *Vector Busy Flag ( VBFlag )* registers consists of 96 1-bit registers ( *VBFlag0 - VBFlag95* ). One flag is assigned to each vector instruction that writes a vector register. These registers are located in VU 66.

At reset, VBFlags are cleared.

25           The *Vector MAP ( VMAP )* structure contains 32 7-bit registers ( *VMAP0 - VMAP31* ) that contain the VBFlag register index which was assigned to the last vector instruction in program order that wrote to the corresponding vector register. VMAP0 will have the VBFlag index that is mapped to V0 and VMAP31 will contain the VBFlag for V31. The VMAP registers are located in the VDU.

At reset, VMAP0 is set to 64, VMAP1 to 65, etc.

The VDU also contains the *Vector Busy Free Queue ( VBFQ )* which consists of 64 7-bit entries that contain the VBFlags that are currently not allocated to any vector instruction pass the VPDL.

At reset, the 64 entries are initialized to 0 - 63.

- 5           For each vector instruction that writes a VR a VBFlag must be mapped before the VPDL can send the vector instruction to the VDQ and/or the VLSDQ. The VPDL will read the current VMAP entry for  $V_i$ , the result register number, push this VBFlag index onto the VBFQ, dequeue the head of the VBFQ, and write this new VBFlag index into the corresponding VMAP entry. The new VBFlag register in VU will be marked
- 10       busy, and the VBFlag index will be passed to the VDQ and/or the VLSDQ with the instruction. When the vector instruction is issued in the VU, the indexed VBFlag will be cleared by the VU. Because the maximum number of vector instructions that can be active after the VPDL is less than 96, there will always be a VBFlag index in the VBFQ.

- For each vector gather/scatter instruction, the VPDL uses the  $V_k$  register number
- 15       to index into the VMAP registers. The current VBFlag index contained in the VMAP table entry is read and passed with the vector instruction to VLSDQ. When the instruction tries to issue in the VLSU, the VBFlag that the index points to is checked to see if it is still set. If the VBFlag is set, the last write of the vector registers has not executed in the VU, so the instruction will hold issue. If the VBFlag is clear, the
- 20       instruction is allowed to be issued if there are no other hold issue conditions.

Once the VPDL has receive notice from the AL and has mapped the VBFlags, it will *vpredispatch* the vector instruction to the VDQ and/or the VLSDQ. All vector instructions are vpredispatched to the VDQ but only vector memory instructions and vector instructions that write VL and VM are sent to the VLSDQ.

- 25           The second stage of the VDU, which performs the last two functions of the VDU, are executed in the *Vector Dispatch Logic ( VDL )* and the *Vector Load/Store Dispatch Logic (VLSDL)*. The VDL and the VLSDL perform the same functions, but execute decoupled from each other.

The VDL assembles the required AU and SU scalar operands for each vector instructions bound for VU 66 and then *vdispatches* these instructions and associated data to the VU. The VLSDL assembles the required AU operands for the vector instructions to be sent to VLSU 68 and *vlsdispatches* these instructions and required data to the VLSU. The VDL and VLSDL uses seven queues to gather the vector instructions and scalar operands:

- 1) The *Vector Dispatch Queue ( VDQ )* can contain 48 valid entries of instructions bound for the VU. The queue can accept one instruction per LClk from the VPDL and the VDL can remove one instruction per LClk from VDQ.
- 2) The *Vector AJoperand Queue ( VAJQ )* contains Aj operands that were read in program order in the AU and sent to the VDU. This queue can contain up to 32 valid entries and is capable of accepting one operand per LClk from the AU.
- 3) The *Vector AKoperand Queue ( VAKQ )* contains Ak operands that were read in program order in the AU and sent to the VDU. This queue can contain up to 32 valid entries and is capable of accepting one operand per LClk from the AU.
- 4) The *Vector SKoperand Queue ( VSKQ )* contains Sk operands that were read in program order in the SU and sent to the VDU. This queue can contain up to 32 valid entries and is capable of accepting one operand per LClk from the SU.
- 5) The *Vector Load/Store Dispatch Queue ( VLSDQ )* can contain 48 valid entries of instructions bound for the VLSU. The queue can accept one instruction per LClk from the VPDL and the VLSDL can remove one instruction per LClk from the VLSDQ.
- 6) The *Vector Load/Store AJoperand Queue ( VLSAJQ )* contains Aj operands that were read in program order in the AU and sent to the VDU. This queue can contain up to 32 valid entries and is capable of accepting one operand per LClk from the AU.
- 7) The *Vector Load/Store AKoperand Queue ( VLSAKQ )* contains Ak operands that were read in program order in the AU and sent to the VDU. This queue can contain up to 32 valid entries and is capable of accepting one operand per LClk from the AU.

All five operand queues can receive data from the AU or SU speculatively. If a branch was mispredicted, a kill signal will be sent from the IFU and all operand data that has been speculatively written into the operand queues after the mispredicted branch will be removed.

5           In operation, the VDL dequeues the head of the VDQ and determine if any operands are required for this vector instruction. If operands are required, the VDL attempts to read the head of the indicated operand queues. If all operands are ready, the VDL *vdispatches* the instruction and data together to the VU.

          In operation, the VLSDL dequeues the head of the VLSDQ and determine if any  
10       operands are required for this vector instruction. If operands are required, the VLSDL attempts to read the head of the indicated operand queues. If all operands are ready, the VLSDL *vlsdispatches* the instruction and data together to the VLSU.

          Order is maintained between the two instructions queues and operand queues by receiving instructions in program order from the VPDL and by also receiving operands  
15       from the AU and SU in program order.

          VU 66 is a two pipe vector unit that is decoupled from SS50 and VLSU 68. VU 66 receives one vector instruction per LClk from VDU 64 in-order and *v issues* these instructions in-order at a maximum of one instruction per LClk. For most vector instructions the VU will then *read* multiple operand elements from a VR file 70, *execute*  
20       multiple vector operation, and *write* multiple result elements to the VR. Different vector instructions can operate out-of-order within VU 66.

          Vector instructions are received from VDU 64 and placed in Vector Instruction Queue (VIQ) 81. VIQ 81 can receive one vector instruction per LClk from VDU 64 and can contain up to six valid instructions. If the vector instruction requires scalar  
25       operands, the operands are dispatched to VU 66 with the vector instruction. If the vector instruction will write a VR, a Vector Busy Flag (VBFlage) index is also sent with the instruction. VDU 64 will not *vdispatch* a vector instruction to VU 66 before all previous scalar instructions have been committed. All vector instructions are sent to VU 66.

Vector Issue Logic (VIL) of VU 66 reads the first valid vector instruction in VIQ 81, resolves all VR conflicts, any functional unit conflicts, and then issues the vector instruction. When a vector instruction is issued, the VIL sets the correct VR busy, busy the appropriate functional unit, and clears the indicated VBFlag if needed.

5 Most vector instructions in the VU are executed in the two Vector Pipes (VP0, VP1).

The 32 64-bit VRs, each 64-elements deep (MaxVL), are split between the two VPs. There are 32 logical and 32 physical VRs with no register renaming. Each VP contains 32 elements of each VR 70 and are stored in RAM in a similar manner to that  
10 used for SR 58 and AR 56.

The 64 elements are split between the VPs by bit 1 of the 6-bit element number. VP0 contains vector elements that have bit 1 set to 0, and VP1 contains vector elements that have bit 1 set to 1. When performing 64-bit operations, the VRs are 64-bits wide and each element is stored in one 64-bit CRR element. When performing 32-bit  
15 operations, the VRs are 32-bit wide and two consecutive elements in a VP are stored in one 64-bit CRR element. The MaxVL for 32-bit operations remains at 64 elements.

This vector register organization has two benefits. One, it allows for 64-bit to 32-bit and 32-bit to 64-bit converts to be executed *within* a VP. Second, it allows 32-bit operations to execute at 4 results per VP per LClk. Because each VR can have a 64-bit  
20 element read *and* written per SClk, with two 32-bit elements packed to one 64-bit element of the CRR, reading 32-bit VR operands can operate at 4 elements per VP per LClk.

In the embodiment of processor 28 shown in Fig. 7, each vector pipe includes three functional unit groups (FUG1, FUG2 and FUG3). The functional unit groups  
25 share common operand and result paths. Since they share paths, only one functional unit in each FUG can be executing at a time.

In one embodiment, FUG1 contains the following vector integer and floating-point functional units: Integer Add/Subtract, Floating-point Add/Subtract, Logical, Integer Compare, and Floating-point Compare. All of FUG1's operations can be either

32-bit or 64-bit operations. FUG1's functional units share two operand paths and one result path. Only one 64-bit operation can be performed in FUG1 per SClk. Two 32-bit operations can be performed per SClk.

5 FUG2 contains the following integer and floating-point functional units: Integer Multiply, Floating-point Multiply, and Shift. All FUG2's operations can be either 32-bit or 64-bit operations. FUG2's functional units also share two operand paths and one result path for the three functional units. It also can perform only one 64-bit or two 32-bit operation per SClk.

FUG3c contains the following integer and floating-point functional units:  
10 Floating-point Divide, Square Root, Leading Zero, Pop Count, Copy Sign, Absolute Value, Logical, Integer Convert, Floating-point Convert, and Merge.

All of FUG3's operations can be either 64-bit or 32-bit operations. FUG3c has two operand paths and one result path. All operations but Divide and Square Root can perform one 64-bit or two 32-bit operations per SClk. FUG3c can perform one 64-bit  
15 Divide or Square Root operation every fourth LClks and 32-bit operations once every two LClks. Both FUG3c for VP0 and VP1 shares one operand path and its result path with two functional units external to the VP0 and VP1 (the Bit Matrix Multiply Unit and the Iota Functional Unit).

FUG4 is a functional unit that performs the following operations on VMs: Fill,  
20 Last element set, First Element Set, Pop Count, and Logical. This functional unit does not read the VRs, but performs single operations by reading and writing the VMs. All these operations are MaxVL -bit operations (64). One FUG4 operation can be issued per LClk.

To control the number of vector operations that are performed for each vector  
25 instruction, the VU contains two other register sets, the *Vector Length Register* ( *VL* ) and eight *Vector Mask Registers* ( *VM* ).

The VL register contains a value between 0 and MaxVL and limits the number of vector elements that are read, vector operations that are performed, and vector

elements that are written when a vector instructions is *vissued* . The 64-bit VL register is set by an AR operand from the AU and can be read back to an AR.

The Vector Mask Registers are used to control vector instructions on a per element basis. They can also contain the results of vector comparison operations. The  
5 eight VMs are each 1-bit wide and MaxVL elements long.

Not all vector instructions are performed under the control of VL and VM registers. Vector instructions that do execute under the control of VL and/or VM registers are called elemental vector instructions

For elemental vector instructions, the current contents of VL are read at *vissue* to  
10 determine the number of operations to perform. At *vissue*, if VL is currently zero, no vector operations are performed. The only elemental vector operations that are not executed under the control of VL are *vi scan*(ak,mj) and *vi cidx*(ak,mj). These two instructions always execute MaxVL elements.

Most elemental vector operations also use a VM register to control the vector  
15 instruction on a per element basis. At *vissue*, the specified VM is used to control which of the VL elements are written. For VM register elements that are set to one, the indicated operations are performed on the corresponding elements. For VM register elements that are cleared, the operations may still be performed but no VR elements will be written, no exception generate, and no data will be written to memory. The bit matrix  
20 multiply (bmm vk) vector instruction is not controlled by VM.

For instructions that are not elemental vector instructions, the number of operations performed is not determined by the contents of the VL and VM registers. Most non-elemental vector operations are memory ordering instructions, single element moves between AR/SRs and the VRs, and VM operations.

25

### **Vector Load/Stores Operations**

Vector memory load and store instructions perform different than the instructions that are executed in the FUGs. Vector memory operations are executed in

both the VU and the VLSU. The VLSU executes decoupled from the VU, generates the addresses for the vector loads and stores, and sends them to EIU 54.

For vector loads, the VU moves the memory data into the VR. For vector stores, the VU reads a VR and sends the store data to the EIU. The store address and data is  
5 sent decoupled to the EIU. This process is described in Patent application No. xx/yyy,yyy, entitled Decoupled Store Address and Data in A Multiprocessor System”, filed herewith, the description of which is incorporated herein by reference.

Vector load and gather instructions are executed using the load buffers 78 of the two VPs. Each VP of VU 66 contains a 256 64-bit entry load buffer (LB) 78 that is  
10 loaded with vector memory data by VLSU 68. In one embodiment, LBs 78 are separated into eight individual load buffers (LBUF0-LBUF7), each 64 64-bit elements deep.

When a vector load instruction reaches the head of the VIQ, the Vector Issue Logic performs two issue checks. First, the vector result register must be not busy. Second, VLSU 68 must have generated all the vector memory addresses without an  
15 address fault, and indicated this to VU 66 by setting the control for one of the eight LBs 78. The VIL will then issue the vector load or gather instruction to the control section of one of the eight LBs. When all the memory load data has been loaded into the LB for this load, the LB control will move the elements from the LB into the VR. Only the elements specified by VL and the VM will be written into the VR. Eight moves from the  
20 LB to the VR can occur per LClk. When all of the elements have been removed from the LB, the LB is marked clear by VU 66 so VLSU 68 can reallocate it. If the addresses generated in the VLSU generated a fault, VU traps.

Vector store and scatter instructions are also executed in both the VU and VLSU. For vector store instructions, the VLSU again generates all of the memory  
25 addresses, transmits them to the EIU, and send notice to the VU if the addresses generated a fault. The VU, then waits for the vector store instruction to reach the head of the VIQ, checks if the VR to be stored is busy, verifies that the addresses generated in the VLSU did not fault and then send the store data to the EIU for the elements that are indicated by the VL and the VM. Each VP has an operand path from the VR to an

external path that is linked to the EIU. This allows for four vector store data operands to be sent per LClk.

VLSU 68 will be discussed next. VLSU 68 is the vector address generation engine of processor 28. As note above, execution within VLSU 68 is decoupled from SS  
5 50 and VU 66.

VLSU 68 generates addresses for VU 66. The VLSU receives its vector instructions from VDU 64, interfaces with VU 66 for gather and scatter index operands, and transmits memory requests to EIU 54.

The VLSU receives one instructions from the VDU per LClk. Instructions are  
10 placed in VLSIQ, which can contain up to six valid instructions. The VLSIQ receives any scalar operands needed for the vector instructions from VDU 64 with the instruction. The VLSU receives all vector memory instructions and any vector instruction that writes a VL or VM register. (VLSU 68 receives instructions that write VL and VM registers because the VLSU maintains its own private copy of the VL and  
15 VM registers.

The Vector Load/Store Issue logic executing in VLSU 68 performs all the required checks for the vector memory instructions at the head of the VLSIQ and attempts to issue the instruction to Vector Address Generation logic. The Vector Load/Store Issue logic can issue one instruction per LClk.

20 The Vector Address Generation logic generates four memory requests per LClk, performs four address translations per LClk using the four Vector TLBs (VTLB), and places the four physical addresses in one of four Vector Request Buffers (VRB0-VRB3).

As noted above, for vector loads, the VLSU assigns one of eight LBs 78. For  
25 strided vector loads, the Vector Load/Store Issue logic only checks for a free LB 78 that can be allocated. If an LB 78 is free, it issues the load to the Vector Address Generation logic, assigns one of the eight LBs 78 to the vector load and informs VU 66 that the selected LB is valid. For vector gather instructions, the Vector Load/Store Issue logic needs to check for an allocatable LBUF and also that the VR needed for the index is

valid and ready. The Vector Load/Store Issue logic uses the VBFlags to check to see that the VR needed for the index is valid and ready.

In one embodiment, VU 66 contains 96 VBFlags. For each vector instruction that writes a VR, VDU 64 assigns a VBFlag index to that instruction and sets the  
5 indexed VBFlag busy. When VU 66 issues the vector instruction, it clears this flag.

When a gather instruction is sent from VDU 64 to VLSU 68, it includes the last VBFlag index of the vector index (Vk). When the gather instruction reaches the head of the VLSIQ, the Vector Load/Store Issue logic checks to see if the indexed VBFlag in the VU is still set. If it is, the last write of the k-operand has not occurred and the Vector  
10 Load/Store Issue logic holds issue. If the VBFlag is cleared, the Vector Load/Store Issue logic then requests the VR from the VU. When the VR is free, (it might still be busy from the instruction that cleared the VBFlag) the VU will send the VR elements to the VLSU four element per LClk. The VU reads these elements from the VR and sends them to the VLSU through an external register path. The Vector Load/Store Issue logic  
15 also assigns an LB 78 to the gather instruction.

In one embodiment, all vector store instructions require verification that all previous scalar and vector loads have been sent to memory. Vector load to store ordering is done by EIU 54 while the scalar load to vector store ordering is enforced by Active List 94 in the DU and by the Vector Load/Store Issue logic. The VLSU receives  
20 a signal from AL 94 each time the "reference sent" pointer passes a vector store instructions. The VLSU can receive four reference sent signals per LClk. The VLSU increments a vector store counter each time it receives a signal. For the Vector Load/Store Issue logic to issue a vector store instruction, the vector store counter must be non-zero. Each time a vector store instruction is issued, the counter is decremented.

25 For scatter operations, the VLSU conducts the same VBFlag busy check that is performed for gather instructions. When the VBFlag is cleared, the VLSU again asks the VU for the VR and the VU send the VLSU four elements per LClk when the register is available.

The Vector Address Generation logic of the VLSU receives the base address from the Vector Load/Store Issue logic. For strided references, it also receive the stride value from the VLSIL. For gather and scatter vector instructions it receives the vector indexes from the VU. The Vector Address Generation logic can generate up to four  
5 addresses per LClk. The number of addresses generated are determined by the contents of VL and the selected VM. If the VL is currently zero, no vector addresses are generated. For non-zero VL, the Vector Address Generation logic generates VL addresses. If the corresponding element in the selected VM is not set, this address is not passed to VTLB.

10 In one embodiment, VLSU 68 contains four VTLBs (VTLB0 - VTLB3). All vector addresses are translated by the VTLBs and perform the required address check. The VTLB reports completion of translation and any error back to the DU. Possible errors are: 1) Address Error: unaligned access, or access to a privileged or reserved memory region; 2) VTLB Fault: miss in the TLB; 3) VTLB Conflict: multiple hits in the  
15 TLB; and 4) VTLB Modify Fault: store to a page marked read only.

After VTLB physical addresses are placed in Vector Request Buffers (VRB0-VRB3). Each VRB can contain 64 valid addresses.

For vector load instructions, the Vector Address Generation logic also creates a Vector Outstanding Request Buffer (VORB) entry for each vector address. The VORB  
20 contains 512 entries that is indexed by the transaction ID (TID) sent to the E chip 24 in the request packet and returned in the response. Each VORB entry contains LB 78 indexes that point to the LB 78 entry that the data should be loaded into when it returns from E Chip 24.

## 25 **Definitions**

In the above discussion, the term “computer” is defined to include any digital or analog data processing unit. Examples include any personal computer, workstation, set top box, mainframe, server, supercomputer, laptop or personal digital assistant capable of embodying the inventions described herein.

Examples of articles comprising computer readable media are floppy disks, hard drives, CD-ROM or DVD media or any other read-write or read-only memory device.

Portions of the above description have been presented in terms of algorithms and symbolic representations of operations on data bits within a computer memory. These  
5 algorithmic descriptions and representations are the ways used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities  
10 take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like. It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate  
15 physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussions, terms such as “processing” or “computing” or “calculating” or “determining” or “displaying” or the like, refer to the action and processes of a computer system, or similar computing device, that manipulates and transforms data represented as physical (e.g., electronic)  
20 quantities within the computer system’s registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

Although specific embodiments have been illustrated and described herein, it will be appreciated by those of ordinary skill in the art that any arrangement which is  
25 calculated to achieve the same purpose may be substituted for the specific embodiment shown. This application is intended to cover any adaptations or variations of the present invention. Therefore, it is intended that this invention be limited only by the claims and the equivalents thereof.